In: Proceedings of the 14th International Conference on Scientific and Statistical Database Management (**SSDBM**)
24th - 26th July 2002, Edinburgh, Scotland, (c) IEEE Computer Society Press

A Cost Model for Interval Intersection Queries on RI-Trees

Hans-Peter Kriegel*, Martin Pfeifle*, Marco Pötke**, Thomas Seidl*
*University of Munich, Institute for Computer Science
{kriegel, pfeifle, seidl}@dbs.informatik.uni-muenchen.de
**sd&m AG software design & management, marco.poetke@sdm.de

Abstract

The efficient management of interval data represents a core requirement for many temporal and spatial database applications. With the Relational Interval Tree (RI-tree[1]), an efficient access method has been proposed to process interval intersection queries on top of existing object-relational database systems. This paper complements that approach by effective and efficient models to estimate the selectivity and the I/O cost of interval intersection queries in order to guide the cost-based optimizer whether and how to include the RI-tree into the execution plan. By design, the models immediately fit to common extensible indexing/ optimization frameworks, and their implementations exploit the built-in statistics facilities of the database server. According to our experimental evaluation on an Oracle database, the average relative error of the estimated cost to the actual cost of index scans ranges from 0% to 23%, depending on the resolution of the persistent statistics and the size of the query objects.

1. Introduction

There is a growing demand for database applications to handle intervals which, for instance, occur as transaction time and valid time ranges in temporal databases [31] [26] [4] or as line segments on a space-filling curve in spatial applications [9] [3]. The SQL:1999 standard provides the datatype PERIOD with the predicates *precedes*, *succeedes*, *meets*, *equals*, *overlaps (= intersects)*, *contains*, and *during* [30]. With the Relational Interval Tree[1] (RI-tree), a relational access method has been proposed which supports all of the PERIOD predicates [18].

Highly accurate but still efficient selectivity estimation and cost prediction are the fundamentals of effective query optimization. As pointed out in [28], standard selectivity estimation does not estimate well the result cardinalities of selections having temporal predicates, and standard built-in methods are not directly suitable for interval intersection queries, in particular. For complex query objects and query predicates, the recent object-relational database servers provide extensible optimization frameworks that come along with the extensible indexing frameworks, in order to complete the seamless integration of user-defined index structures into the declarative DML. As an example for such an extension, we propose a cost model for the RI-tree that fits well to the extensible frameworks by design. Though the RI-tree immediately maps intervals to built-in B+-trees, the built-in cost models for B+-trees do not estimate well the processing cost since they do not take the particular structure and partitioning of interval data into account.

Our techniques aim at the collection of statistics, the estimation of selectivity, and the prediction of I/O cost. Thereby, the optimizer of the database system is enabled to place the user-defined index at its optimal position in the query execution plan. According to [5] and [13], such a cost-based approach is preferable to rule-based approaches when referencing user-defined methods as predicates. The two main design aspects for the above mentioned functions are:

Effectiveness. The extensible optimizer uses the selectivity estimation to determine a good join order for complex SQL queries. It then evaluates the available cost models to choose the most efficient access path to the data. The objective is to keep the relative error of selectivity and cost estimations sufficiently small to rank the user-defined index accurately among alternative access methods.

Efficiency. In order to obtain an efficient execution plan for a DML operation, the optimizer framework calls the estimation functions for each contained interval predicate. To reduce the total runtime of query optimization, the execution cost for the estimation functions should be kept minimal. Furthermore, data statistics required for the estimation functions should also be efficiently collected.

The architecture of extensible optimization is analogous to extensible indexing as illustrated in Figure 1: Whereas the new methods are built on top of the relational SQL layer, they are object-relationally embedded by implementing the

1. Patent pending.

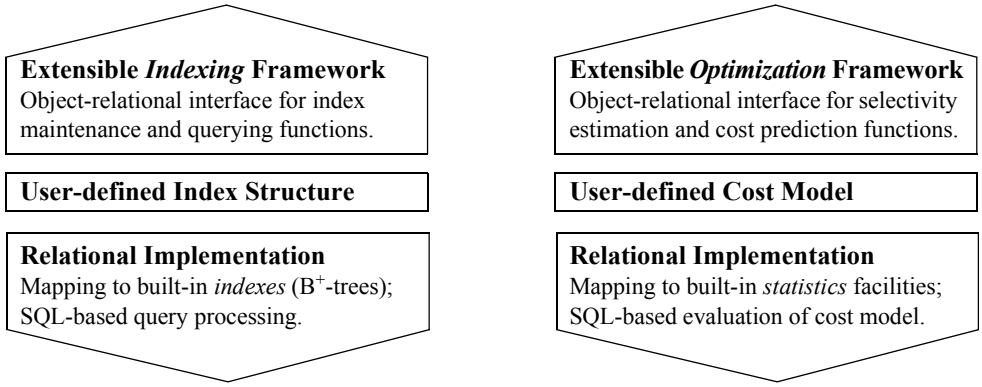

Fig. 1. Analoguous architectures for the object-relational embedding of user-defined index structures and cost models into extensible indexing and optimization frameworks, respectively.

respective interfaces of the frameworks. In case of our new cost models, we particularly propose methods to estimate the selectivity of a given range query on a database of intervals (function *getSelectivity*) and a method to predict the cost of processing that query (function *getIndexCost*). In this paper, we focus on the predicate *overlaps* (= *intersects*) which is considered to be the most important one [10]. Nevertheless, the RI-tree supports all of the PERIOD predicates as well [18] and, moreover, has been also successfully applied to spatial queries [19]. The proposed cost model can easily be extended to these kinds of queries.

The organization of the paper follows the requirements of extensible optimization frameworks and proceeds in the following way: After sketching related work on selectivity estimation and cost prediction in Section 2, Section 3 briefly recalls the RI-tree [17]. In Section 4, we propose two approaches to estimate the selectivity of intersection queries on interval data. The first approach is based on user-defined histograms, whereas the second one relies on the built-in statistics of standard database systems. Section 5 derives a cost model for estimating the I/O cost of a given query on the RI-tree. After an empirical evaluation of the presented methods in Section 6, the paper is concluded in Section 7.

2. Related Work

2.1. Selectivity Estimation

In order to determine a good estimate for the selectivity of a specific predicate without retrieving the actual results, the predicate has to be evaluated on a sufficiently accurate approximation of the data distribution. The computation of such an approximation is known as one of the most difficult problems, for instance in case of selectivity estimation of extended objects [1]. The many existing approaches fall into three different classes: parametric techniques, sampling, and statistics.

Parametric techniques approximate the given data by using a standard mathematical distribution. For databases comprising extended objects, many proposals exploit intrinsic characteristics of the stored data, including the usage of the *Correlation Fractal Dimension* on point sets by Belussi and Faloutsos [2] or the *SLED* property of real segment data proposed by Proietti and Faloutsos [24]. A limitation for parametric techniques results from the requirement of a-priori assumptions about the data distribution. In contrast, *sampling* adapts to the actual data distribution by processing a small fraction of the stored tuples. This paradigm has been pursued and evaluated by Lipton, Naughton and Schneider [21] and by Haas et al. [12]. *Statistics* are a very popular approach in database systems, as they typically can be efficiently computed and occupy only a small amount of secondary storage. For linearly ordered domains, the most commonly used statistics type in commercial database servers are quantiles of the original data. For the selectivity estimation on non-uniform distributions of extended objects, histograms are a common technique. An extensive analysis on different kinds of spatial histograms has been published by Acharya, Poosala and Ramaswamy [1]. Whereas histograms can be naturally applied to one-dimensional interval data, a quantile-based approach has to operate on a linear representation of the original intervals. In this paper, we present and evaluate techniques for interval data on both types of statistics, histograms as well as quantiles.

2.2. Cost Estimation

A wide range of cost models has been presented in the literature for various index structures for extended objects, including the technique of Kamel and Faloutsos [16] for intersection queries on packed R-trees, or the *REGAL* law for R-tree entries by Proietti and Faloutsos [25]. Recently, cost models have also been extended to handle joins of extended

objects and the presence of database buffers as in the proposals of Huang, Jing and Rundensteiner [11], Leutenegger and Lopez [20], or Theodoridis, Stefanakis and Sellis [33]. Whereas previous research has mainly concentrated on the design and evaluation of cost models for stand-alone access methods, the following sections develop an approach that can be fully implemented on top of existing object-relational database systems.

3. The Relational Interval Tree

The RI-tree is a relational storage structure for interval data (*lower*, *upper*), built on top of the SQL layer of any RDBS. By design, it follows the concept of Edelsbrunner's main-memory interval tree [8] and guarantees the optimal complexity for storage space and for I/O operations when updating or querying large sets of intervals.

3.1. Relational Storage and Extensible Indexing

The RI-tree strictly follows the paradigm of relational storage structures since its implementation is purely built on (procedural and declarative) SQL but does not assume any lower level interfaces to the database system. In particular, built-in index structures are used as they are, and no intrusive augmentation or modification of the database kernel is required.

On top of its pure relational implementation, the RI-tree is ready for immediate object-relational wrapping. It fits particularly well to extensible indexing frameworks as already proposed in [32] and illustrated in Figure 1. These frameworks, which are provided by the latest object-relational database systems, including IBM DB2 Universal Database [14] [7], Informix Universal Server [15] [6], or Oracle Server [23] [29] enable developers to extend the set of built-in index structures by custom access methods in order to support user-defined datatypes and predicates without weakening the reliability of the entire system.

3.2. Dynamic Data Structure

The structure of an RI-tree consists of a binary tree of height h which covers the range $[1, 2^h–1]$ of potential interval bounds. It is called the virtual backbone of the RI-tree since it is not materialized but only the root value $2^{h–1}$ is stored persistently in a metadata table. Traversals of the virtual backbone are performed purely arithmetically by starting at the root value and proceeding in positive or negative steps of decreasing length $2^{h–i}$, thus reaching any desired value of the data space in $O(h)$ CPU time and without causing any I/O operation. For the relational storage of intervals, the node values of the tree are used as artificial keys: Upon insertion of an interval, the first node that hits the interval when descending the tree from the root node down to the interval location is assigned to that interval.

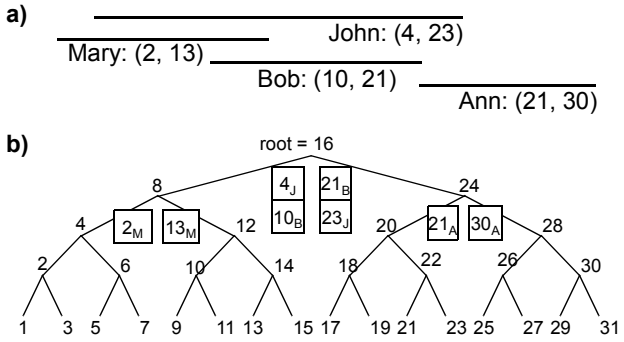

Fig. 2. Example for an RI-tree. **a)** four intervals. **b)** virtual backbone and registration positions of the intervals. **c)** resulting relational indexes *lowerIndex* and *upperIndex*

An instance of the RI-tree then consists of two relational indexes which in an extensible indexing environment are preferably managed as index-organized tables. The indexes obey the relational schema *lowerIndex* (*node, lower, id*) and *upperIndex* (*node, upper, id*) and store the artificial key value *node*, the bounds *lower* and *upper*, and the *id* of each interval. An interval is represented by exactly one entry in each of the two indexes, and therefore, $O(n/b)$ disk blocks of size b suffice to store n intervals. For inserting or deleting intervals, the *node* values are determined arithmetically, and updating the indexes requires $O(\log_b n)$ I/O operations per interval.

The illustration in Figure 2 provides an example for the RI-tree. Let us assume the intervals (2,13) for Mary, (4,23) for John, (10,21) for Bob, and (21,30) for Ann (Fig. 2a). The virtual backbone is rooted at 16 and covers the data space from 1 to 31 (Fig. 2b). The intervals are registered at the nodes 8, 16, and 24. The interval (2,13) for Mary is represented by the entries (8, 2, Mary) in the *lowerIndex* and (8, 13, Mary) in the *upperIndex* since 8 is the registration node, and 2 and 13 are the lower and upper bound, respectively (Fig. 2c).

3.3. Intersection Query Processing

To minimize barrier crossings between the procedural runtime environment and the declarative SQL layer, an interval intersection query (*lower*, *upper*) is processed in two steps. The procedural query preparation step descends the

virtual backbone from the root node down to *lower* and to *upper*, respectively. The traversal is performed arithmetically without causing any I/O operations, and the visited nodes are collected in two main-memory tables, *left queries* and *right queries*, as follows: nodes to the left of *lower* may contain intervals which overlap *lower* and are inserted into *left queries*. Analogously, nodes to the right of *upper* may contain intervals which overlap *upper* and are inserted into *right queries*. Whereas these nodes are taken from the paths, the set of all nodes between *lower* and *upper* belongs to the so-called *inner query* which is represented by a single range query on the node values. All intervals registered at nodes from the *inner query* are guaranteed to intersect the query and, therefore, will be reported without any further comparison. The query preparation step is purely based on main memory and requires no I/O operations.

In the subsequent declarative query processing step, the transient tables are joined with the relational indexes *upperIndex* and *lowerIndex* by a single, three-fold SQL statement (Figure 3). The upper bound of each interval registered at nodes in *left queries* is compared to *lower*, and the lower bounds of intervals stemming from *right queries* are compared to *upper*. The *inner query* corresponds to a simple range scan over the intervals with nodes in (*lower*, *upper*). The SQL query requires $O(h \cdot \log_b n + r/b)$ I/Os to report r results from an RI-tree of height h since the output from the relational indexes is fully blocked for each join partner.

```
SELECT id FROM upperIndex i, :leftQueries q
    WHERE i.node = q.node AND i.upper >= :lower
UNION ALL
SELECT id FROM lowerIndex i, :rightQueries q
    WHERE i.node = q.node AND i.lower <= :upper
UNION ALL
SELECT id FROM lowerIndex   /* or upperIndex */
    WHERE node BETWEEN :lower AND :upper;
```

Fig. 3. SQL statement for an intersection query with bind variables for *left queries*, *right queries*, *lower* and *upper*

For systems which do not support transient main-memory tables as bind variables, we use set containment predicates 'i.node IN leftQueries' and 'i.node IN rightQueries'. The query sets are then composed by string concatenation. In any case, no I/O operations are required in the query preparation step.

4. Selectivity Estimation

Accurate estimations of query result sizes are a necessary input for many components of the underlying database system. In particular, the selectivity estimation for an interval intersection query can be used by the built-in optimizer

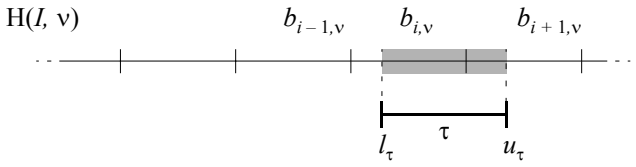

Fig. 4. Selectivity estimation on an interval histogram.

to find an efficient join order and to determine the best available access method [27] [23]. Selectivity estimation is also required to provide the user with an approximate prediction about the potential execution time of DML statements. In the following, we propose a histogram-based approach ('equi-width histograms') and a quantile-based approach ('equi-count histograms').

4.1. Histogram-Based Selectivity Estimation

In order to cope with arbitrary interval distributions, histograms can be employed to capture the data characteristics at any desired resolution. We start by giving the definition of an interval histogram:

Definition 1 *(Interval Histogram)*.
Let $D = [1, 2^h - 1]$ be a domain of interval bounds, $h \geq 1$. Let the natural number $\nu \in \mathbb{N}$ be the *resolution*, and $\beta_\nu = (2^h - 1)/\nu$ the corresponding *bucket size*. Let $b_{i,\nu} = [1 + (i - 1)\cdot\beta_\nu, 1 + i\cdot\beta_\nu)$ denote the *span of bucket* i, $i \in \{1, \ldots, \nu\}$. Let further $I = \{(l, u), l \leq u\} \subseteq D^2$ be a database of intervals. Then, $H(I, \nu) = (n_1, \ldots, n_\nu) \in \mathbb{N}^\nu$ is called the *interval histogram* on I with *resolution* ν, iff for all $i \in \{1, \ldots, \nu\}$:

$$n_i = |\{\psi \in I \mid \psi \text{ intersects } b_{i,\nu}\}|$$

In order to compute an interval histogram on a database I of n intervals, $O(n/b)$ disk blocks have to be touched, assuming a blocked storage of I by a page size b. The computation is performed by standard SQL and wrapped by a stored procedure that complies with the statistics collection interface of the extensible optimization framework (function *getSelectivity*). Based on $H(I, \nu)$, we compute a selectivity estimate by evaluating the intersection of the query interval with each bucket span $b_{i,\nu}$ (cf. Figure 4):

Definition 2 *(Histogram-based Selectivity Estimate)*.
Given an interval histogram $H(I, \nu) = (n_1, \ldots, n_\nu)$ with bucket size β, we define the *histogram-based selectivity estimate* $\sigma_I(I, \tau)$, $0 \leq \sigma_I(I, \tau) \leq 1$ for an intersection query $\tau = (l_\tau, u_\tau)$ by the following formula:

$$\sigma_I(I, \tau) = \left[\sum_{i=1}^{\nu} \frac{overlap(\tau, b_{i,\nu})}{\beta} \cdot n_i\right] \cdot \left[\sum_{i=1}^{\nu} n_i\right]^{-1}$$

where *overlap* returns the intersection length of two intersecting intervals, and 0, if the intervals are disjoint.

Note that long intervals may span multiple histogram buckets. Thus, in the above computation, we normalize the expected output to the sum of the number n_i of intervals intersecting each bucket i rather than to the original cardinality n of the database. In order to support query intervals with a very small duration, the average length of the stored intervals could also be considered for the estimation.

4.2. Quantile-Based Selectivity Estimation

Due to the replication of intervals across bucket boundaries, the accuracy of the histogram-based selectivity estimation may deteriorate with longer interval lengths or higher histogram resolutions. In addition, the runtime required for the histogram computation is increased by the cost of barrier-crossings between the declarative environment of the SQL layer and our stored procedure. Fortunately, most ORDBMS comprise efficient built-in functions to compute single-column statistics, particularly for cost-based query optimization. Available optimizer statistics are accessible to the user by the relational data dictionary. The basic idea of our quantile-based selectivity estimation is to exploit these built-in index statistics rather than to add and maintain user-defined histograms. We start with the definition of a quantile vector, the typical statistics type supported by relational database kernels. Then, we describe its application to node values of the RI-tree.

Definition 3 *(Quantile Vector)*.
Let $(S, \leq)$ be a totally ordered multi-set. Without loss of generality, let $S = \{s_1, s_2, \ldots, s_k\}$ with $s_j \leq s_{j+1}$, $1 \leq j < k$. Then, $Q(S, \nu) = (q_0, \ldots, q_\nu) \in S^\nu$ is called a *quantile vector* for S and a *resolution* $\nu \in \mathbb{N}$, iff the following conditions hold:

(*i*) $q_0 = s_1$

(*ii*) $\forall i \in 1, \ldots, \nu: \exists j \in 1, \ldots, k: \quad q_i = s_j \wedge \frac{j-1}{k} < \frac{i}{\nu} \leq \frac{j}{k}$

Definition 4 *(Node Quantiles)*.
Let *lowerIndex* be the relational index on (*node*, *lower*, *id*) for an instance T of the RI-tree. Let $N = \pi_{node}(lowerIndex)$ be the projected multi-set of node values. Then, $Q(N, \nu)$ is called the vector of *node quantiles* on T with *resolution* ν.

Based on the node ordering materialized in the *lowerIndex* (or *upperIndex*), the computation of $Q(N, \nu)$ on an RI-tree storing n intervals has an I/O complexity of $O(n/b)$, where b is the disk block size. By using the node quantiles for an RI-tree index, we get an aggregated view on the locations of the stored intervals. In addition, we may use some knowledge about the one-dimensional durations which is given by the following definition that captures the average distances of the interval bounds to the respective fork node:

Definition 5 *(Average Node Distances)*.
Let T be an instance of the RI-tree with *lowerIndex* and *upperIndex* relations. Then, the *average lower distance* $\delta_{lower}(T)$ and the *average upper distance* $\delta_{upper}(T)$ is defined as:

(*i*) $\delta_{lower}(T) = \text{avg}(node - lower)$

(*ii*) $\delta_{upper}(T) = \text{avg}(upper - node)$

The average values δ_{lower} and δ_{upper} are computed with $O(n/b)$ I/O complexity, and if possible, along with the quantile statistics. If the built-in statistics of the hosting database system comprise single-column averages on *node*, *lower*, and *upper*, then δ_{lower} and δ_{upper} can be simply derived from these existing statistics: $\delta_{lower} = \text{avg}(node) - \text{avg}(lower)$ and $\delta_{upper} = \text{avg}(upper) - \text{avg}(node)$. For interval databases with a highly skewed distribution of interval lengths, δ_{lower} and δ_{upper} can be replaced by quantiles on $\pi_{node-lower}(lowerIndex)$ and $\pi_{upper-node}(upperIndex)$.

Our goal is to compute the selectivity estimate in constant time, i.e. independent not only from the cardinality, but also from the granularity of the interval data. Instead of submitting the $O(h = \log_2 root + 1)$ node queries on the RI-tree, we evaluate the quantiles with respect to the span of nodes touched during the processing of a potential interval intersection query.

Definition 6 *(Span of Touched Nodes)*.
For a given RI-tree T and an intersection query τ, the range $\theta(T, \tau) = (l_\theta, u_\theta)$ is called the *span of touched nodes*, iff l_θ is the minimal and u_θ is the maximal node on the virtual backbone that is touched while processing the query τ on T.

Lemma 1. *Let $D = [1, 2^h - 1]$ be the interval domain covered by an RI-tree T with root $= 2^{h-1}$. For an intersection query $\tau = (l_\tau, u_\tau) \in D$, the span of touched nodes $\theta(T, \tau) = (l_\theta, u_\theta) \in D$ is computed by the following formulas:*

(*i*) $l_\theta = 2^k, k = \log_2(l_\tau)$,

(*ii*) $u_\theta = 2^h - 2^k, k = \lfloor \log_2(2^h - u_\tau) \rfloor$.

Proof. (*i*) The leftmost node touched during the arithmetic traversal of the backbone is the last node before we first step into a right subtree. Following the left branch yields a 0-bit, following the right branch yields a 1-bit in the binary representation of the actual node value. Thus, the leftmost node l_θ has exactly one bit set at the first position of a 1-bit in l_τ. (*ii*) Analogously, the rightmost node u_θ is derived from the first 0-bit in the binary representation of u_τ by a mirrored consideration. ■

We estimate the number of results yielded by the *inner*, *left*, and *right queries* for an intersection query $\tau = (l_\tau, u_\tau)$ based on the node quantiles $Q(N, \nu) = (q_0, \ldots, q_\nu)$. Figure 5 provides a graphical interpretation of the following calculations: the number of results r_{inner} from the *inner query* can

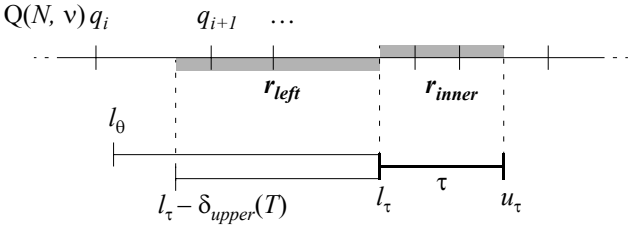

Fig. 5. Selectivity estimation on node quantiles.

be estimated by evaluating the overlap of τ with the quantiles (analogously to Section 4.1):

$$r_{inner} = \sum_{i=1}^{\nu} \left(\frac{overlap(\tau, (q_{i-1}, q_i))}{q_i - q_{i-1}} \cdot \frac{|N|}{\nu} \right)$$

To estimate the number of results r_{left} retrieved by the *left queries*, we only have to consider quantiles falling into the range $(leftBound_\tau, l_\tau)$, where $leftBound_\tau = \max(l_\theta, l_\tau - \delta_{upper}(T))$ and $\theta(T, \tau) = (l_\theta, u_\theta)$:

$$r_{left} = \sum_{i=1}^{\nu} \left(\frac{overlap((leftBound_\tau, l_\tau), (q_{i-1}, q_i))}{q_i - q_{i-1}} \cdot \frac{|N|}{\nu} \right)$$

The estimation of the number of results r_{right} of the *right queries* is done analogously to r_{left}. Finally, we define:

Definition 7 *(Quantile-based Selectivity Estimate)*. The quantile-based selectivity estimate $\sigma_N(I, \tau)$ of the intersection query τ on an interval database I is given by

$$\sigma_N(I, \tau) = \frac{r_{left} + r_{inner} + r_{right}}{|N|}$$

As desired, the quantile vector is a non-replicating statistics on interval data, and the data sets contributing to the results r_{left}, r_{inner}, and r_{right} are disjoint. In consequence, $0 \le r_{left} + r_{inner} + r_{right} \le |N|$ holds and, thus, $0 \le \sigma_N(I, \tau) \le 1$.

5. Model for I/O Cost

In order to achieve a seamless declarative integration of the Relational Interval Tree into extensible indexing frameworks as provided by modern object-relational database systems, a cost model has to be registered at the extensible optimization framework. In this section, we present a cost model for interval intersection queries on the RI-tree, based on the estimated selectivity and the range queries generated for the underlying B$^+$-trees.

We assume the selectivity estimation $\sigma(I, \tau)$ for an intersection query $\tau = (l_\tau, u_\tau)$ on an interval data set I to be determined as shown above. In our derivation of a cost model to estimate the number of touched B$^+$-tree blocks for arbitrary intersection queries τ, we use that expected selectivity as input for the estimation of the I/O operations.

Let us recall from Section 3.3 that the query preparation step does actually cause no I/O operations since the traversal of the backbone structure is done purely arithmetically, and the generated join partners are managed in main memory. The I/O complexity of $O(h \cdot \log_b n + r/b)$ [17] for an intersection query retrieving r results from an RI-tree of height h comprises components of the following two types:

- First, the directories of the relational indexes (built-in B$^+$-trees) have to be traversed in order to navigate on the disk to the first result, if any, for each join partner. Let us denote this portion of I/O operations by $join_{I/O}$ and let us recall that $join_{I/O} = O(h \cdot \log_b n)$.
- Second, the results for each join partner are reported by scanning contiguous leaf blocks of the relational indexes. We call this portion of I/O operations $output_{I/O}$. Since the output is blocked, i.e. there are no gaps between the answers for a single range query, the complexity $output_{I/O} = O(r/b)$ is guaranteed.

In contrast to the very general complexity analysis, a cost model has to compute actual numbers of I/O operations for specific interval queries. Our model relies on the following two observations:

1. In a real user environment with many concurrent queries, substantial parts of the B$^+$-directories typically reside in the main memory and can be managed by the built-in LRU-cache of the DBMS [22]. According to a common assumption, we count two I/O operations for each leaf-block access in order to estimate the number of blocks actually read from disk.
2. The transient join partners are processed in increasing order (*left queries*, *inner query*) or decreasing order (*right queries*) with respect to the *node* value in the composite indexes on (*node*, *upper, id*) and (*node*, *lower, id*), respectively. Due to this ordered access, pages that are read several times during query processing will rarely be displaced from the LRU cache between the accesses. We therefore assume that each leaf page is retrieved only once from secondary storage.

Based on these assumptions, we derive individual formulas for the components $output_{I/O}$ and $join_{I/O}$ in the following.

***output*$_{I/O}$.** For a given RI-Tree T on a set I of intervals, let L = leaf-blocks(*upperIndex*) ≈ leaf-blocks(*lowerIndex*) be the number of leaf blocks in the B$^+$-trees, $L = O(n/b)$, and τ be an interval intersection query performed on T. The answers retrieved from *upperIndex* and from *lowerIndex* are guaranteed to be disjoint, and we estimate $output_{I/O}(T, \tau)$ as the fraction of L predicted by the selectivity estimate $\sigma(I, \tau)$ on T:

$$output_{I/O}(T, \tau) = \sigma(I, \tau) \cdot L$$

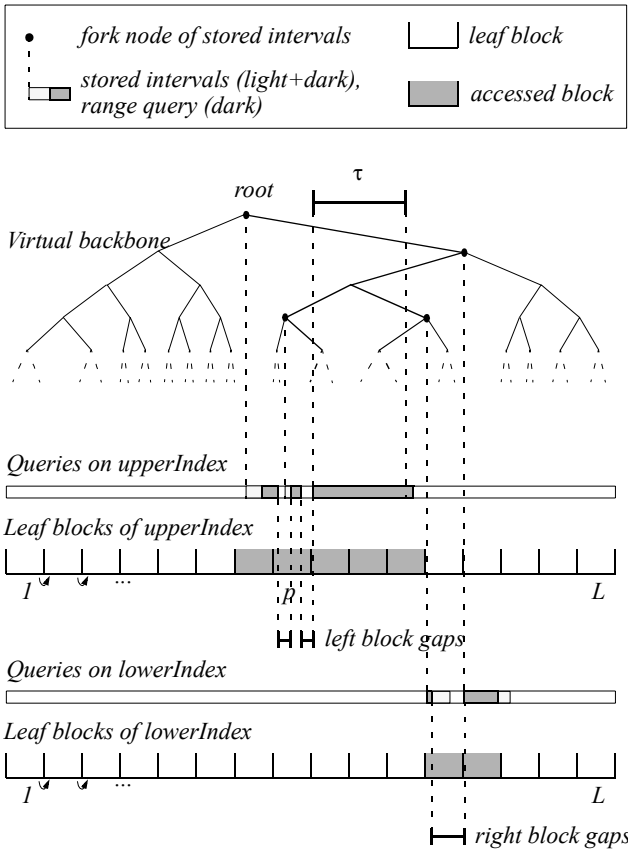

Fig. 6. Touched leaf blocks and query gaps for an intersection query τ.

***join**$_{I/O}$.* The formula for $join_{I/O}$ includes the number of leaf block accesses caused by the navigation in the B$^+$-tree directories for the join partners. Since the leaf blocks are read from two independent B$^+$-tree indexes, we capture the join overhead for the set of *left queries* and *inner queries* on the one hand and for the set of *right queries* on the other hand separately.

Figure 6 provides an illustration for our considerations and depicts the leaf blocks in the *lowerIndex* and *upperIndex* that are read for a query $\tau = (l_\tau, u_\tau)$. Note that the virtual backbone is drawn to the scale of the population in the indexes, and not to the original domain of $D = [1, 2^h - 1]$.

The leaf block p in the *upperIndex*, for example, is touched multiple times during query processing. According to the locality-preserving read schedules for LRU buffers, the multiple accesses to block p count for a single leaf access only. This estimation is complemented by the additional heuristics to count two physical disk accesses for a single leaf block access in order to take care of the I/O caused by traversing the index directory.

An important observation for $join_{I/O}$ is that the results of different join partners in general do not form a contiguous

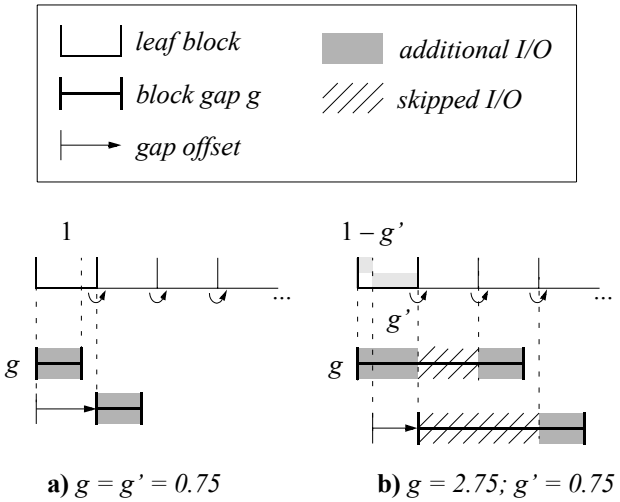

Fig. 7. Additional I/O due to block gaps *g* between range queries.

range of entries in the leaf blocks of the indexes. Although the results are blocked for each single *left query*, *inner query*, and *right query*, there are typically gaps between the blocked result sets of different join partners. In order to model the distribution of gaps, we first determine the gaps between the node values, $NGaps_{left}(\tau)$ and $NGaps_{right}(\tau)$, for a given intersection query τ on an RI-tree T. Then, we derive the expected corresponding gaps between disk blocks, $BGaps_{left}(\tau)$ and $BGaps_{right}(\tau)$.

Estimation of Node Gaps. For the estimation of node gaps, we traverse the virtual backbone on $D = [1, 2^h - 1]$, and we collect the lengths $NGaps_{left}(\tau) = \{\zeta_1, \ldots, \zeta_l\}$ of gaps to the left of the query interval τ, i.e. in the range $[1, l_\tau]$ between consecutive nodes touched by the *left* and *inner queries*, and the lengths $NGaps_{right}(\tau) = \{\xi_1, \ldots, \xi_r\}$ of gaps to the right of τ, i.e. in the range $[u_\tau, 2^h - 1]$ between the *right queries*, respectively.

Estimation of Block Gaps. Let L_θ be the average number of nodes per leaf block in the span $\theta(T, \tau)$ of touched nodes for τ. We estimate the corresponding block gaps among the range queries for τ by the multi-sets $BGaps_{left}(\tau)$ and $BGaps_{right}(\tau)$ of real numbers:

$$BGaps_{left}(\tau) = \frac{\zeta_1}{L_\theta}, \ldots, \frac{\zeta_l}{L_\theta}, \; BGaps_{right}(\tau) = \frac{\xi_1}{L_\theta}, \ldots, \frac{\xi_r}{L_\theta}.$$

The value of L_θ is easily estimated by using the persistent statistics on T along with the cardinality n and the number of leaf blocks L, similarly to Section 4. After having computed the number and extension of gaps between the blocked sections of $output_{I/O}$, we use this information to estimate $join_{I/O}$. Depending on the length and the position of each block gap g, a specific number of leaf block accesses

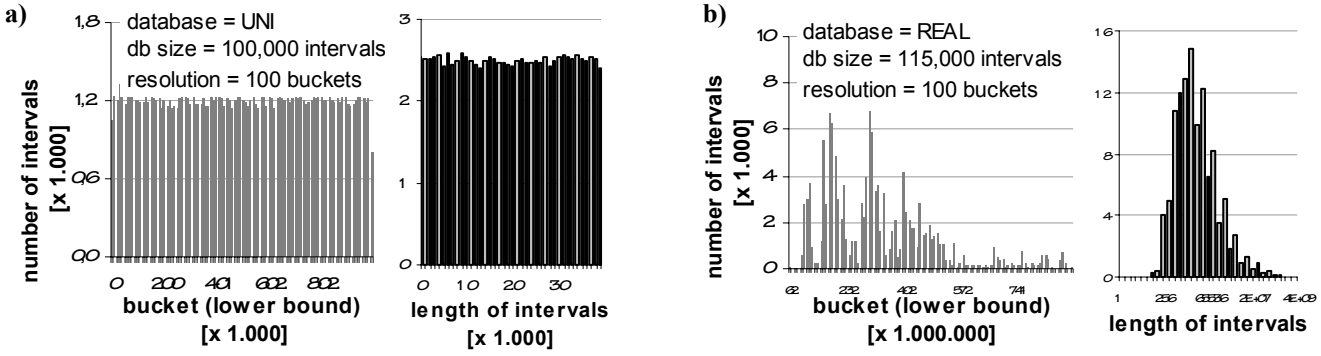

Fig. 8. Histograms of interval distributions **a)** for uniform data, **b)** for real data.

occurs. For gaps smaller than one disk block, i.e. $g \leq 1$, the I/O is increased by this very gap length g with a weight of 1 (cf. Figure 7a). According to Figure 7b, the I/O overhead for larger gaps depends on the gap offset to the leaf blocks and is restricted to blocks at the gap border. For gaps $g > 1$, our formula to estimate the contribution $gap_{I/O}(g)$ of a gap g to $join_{I/O}$ therefore focuses on the fraction $g' = g - \lfloor g \rfloor$. Since we assume a uniform distribution of gap offsets with respect to the leaf blocks in the *upperIndex* and *lowerIndex*, the mean contributions of the left and right borders of a gap $g > 1$ to $gap_{I/O}(g)$ are $1 + g'$ with weight $1 - g'$ and g' with weight g'. The overall value then sums to $(1 - g') \cdot (1 + g') + g' \cdot g' = 1$, and the distinction of cases simplifies to

$$gap_{I/O}(g) = \min(1, g).$$

With respect to I/O cost, random access to a leaf block is, therefore, only beneficial if the preceding block gap is larger than the size of a disk block. In consequence, gaps covering only fractions of a disk block could be sequentially scanned without causing any I/O overhead (this observation opens up a promising potential to further improve the performance of the RI-tree). For all gaps between the range queries on the *upperIndex* and *lowerIndex* for a given query interval τ on an RI-tree T, we estimate the additional I/O for the join processing as

$$join_{I/O}(T, \tau) = \sum_{g \in BGaps_{left}(\tau) \cup BGaps_{right}(\tau)} gap_{I/O}(g).$$

The total I/O cost for an interval intersection query τ on an RI-tree T is then summarized by

$$total\_cost_{I/O}(T, \tau) = output_{I/O}(T, \tau) + join_{I/O}(T, \tau).$$

6. Empirical Evaluation

6.1. Experimental Setup

We implemented the proposed functions for the estimation of selectivity and execution cost on the Oracle Server Release 8.1.6 using built-in methods for statistics collection, analytic SQL functions, and the PL/SQL procedural runtime environment. All experiments were performed on an Athlon/750 machine with IDE hard drives. The database block cache was set to 500 disk blocks with a block size of 8 KB and was used exclusively by one active session. The experiments for the evaluation of statistics, selectivity estimation, and cost model have been executed on various interval databases. We have used a synthetic dataset of intervals following a uniform starting point and length distribution (*UNI*) and intervals derived from a real dataset (*REAL*). For both databases UNI and REAL, Figure 8 depicts the histogram statistics. A peak in the histogram visualization denotes a high density of interval data. In case of the UNI dataset, the data space [1..1,000,000] is covered with a uniform density.

To evaluate the quality of the selectivity and cost prediction, we determined the average relative error of the estimates. This measure denotes the ratio of the absolute estimation error to the actual query result, averaged over a set of queries S. If e_i is the estimated and r_i is the actual result size of a query q_i, the average relative error of the estimated selectivity for S is defined as:

$$\text{Avg relative error (selectivity)} = \left(\sum_{q_i \in S} |r_i - e_i| \right) / \left(\sum_{q_i \in S} r_i \right)$$

For the estimations of the actual I/O cost, the average relative error is defined analogously. This measure is a common technique to evaluate selectivity estimations and cost models, see e.g. [1]. It is undefined if all queries in a query set produce zero output or zero cost. However, this is not the case for our evaluation. As an alternative, the geometric average of relative errors could be used as in [25]. But, as the relative error of some predictions reached zero, this measure would be undefined, and, therefore, could not be used throughout our experiments. The following results show the

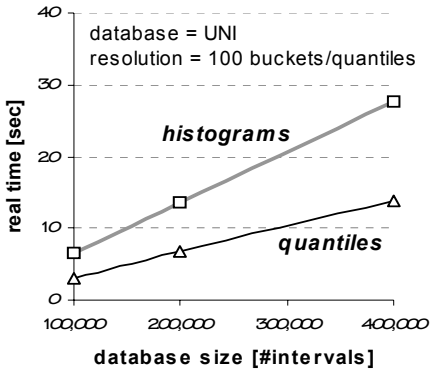

Fig. 9. Computation cost of histogram-based and quantile-based statistics.

averages of, in total, 100 intersection queries for the UNI and REAL databases.

6.2. Computation of Statistics

The persistent statistics must be recomputed in order to adapt to changing data distributions. For highly dynamic data, the database administrator might even decide to trigger the computation of important statistics periodically. Therefore, a low execution cost for the creation of statistics is essential. Figure 9 compares the total runtime of computation for the histogram statistics to the quantile statistics for increasing database size, using 100% samples. Due to the overhead of barrier crossing between PL/SQL and SQL, the quantile-based approach outperforms the histogram-based approach by a factor of 2.

6.3. Selectivity Estimation

In the next set of experiments, we evaluate the average relative error with respect to the query size, i.e. the percentage of the data space covered by the query region. Figure 10 shows the relative error of the histogram-based and quantile-based statistics on the UNI and REAL database. The resulting accuracy of both, the quantile-based approach and the histogram-based approach is very high. For higher selectivities, the quantile-based approach performs slightly better, yielding estimation errors around 4.5% and 2.9% for the UNI and REAL database, respectively. This result can be explained by the fact that quantiles adapt to the local density of the data, whereas histograms partition the whole data space using buckets of identical size. The next experiment in Figure 11 depicts the average relative error for different resolutions of the persistent statistics, evaluated for a set of intersection queries having 10% average query size. As expected, the estimation error increases significantly for coarser resolutions due to the replication of intervals across

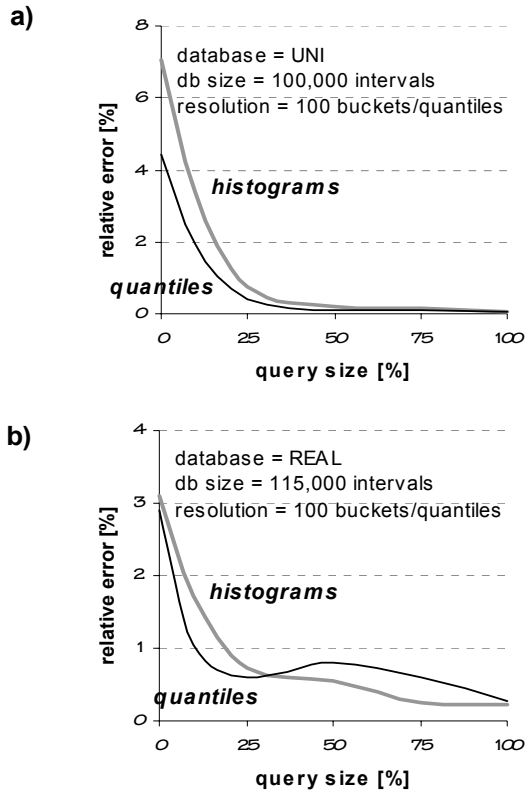

Fig. 10. Relative error of selectivity estimation for histogram-based and quantile-based statistics **a)** on uniform data, **b)** on real data.

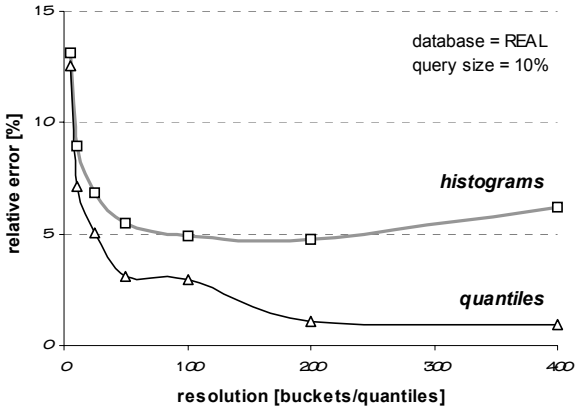

Fig. 11. Relative error of selectivity estimation for varying resolution of the statistics.

bucket boundaries (cf. Section 4.2). Beyond a global optimum at some 100 buckets, the error of the histogram-based approach increases for higher resolutions, due to the repli-

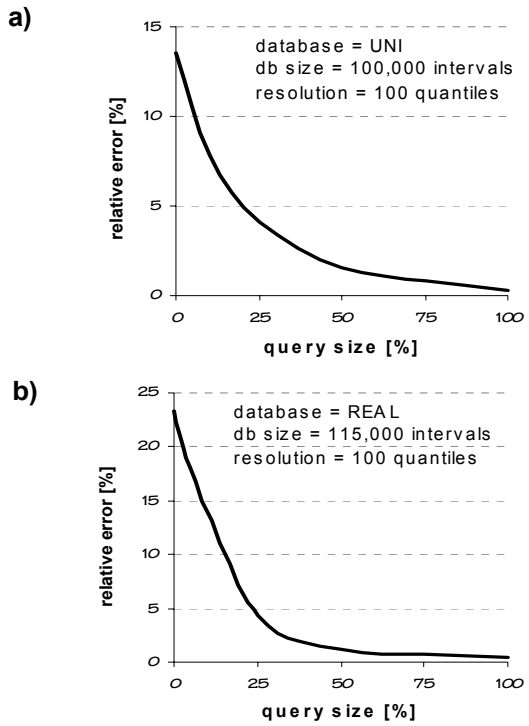

Fig. 12. Relative error for cost estimation **a)** on uniform data, **b)** on real data.

cation of intervals spanning multiple histogram buckets. Therefore, we focus on the quantile-based approach in the following experiments, as the representation of intervals is non-redundant. Regarding the runtime, a single selectivity estimation using statistics with a resolution of 100 quantiles for the UNI and REAL databases took about 0.05 seconds on the average.

6.4. Cost Estimation

We used the estimated quantile-based selectivity of the previous section as input for the I/O cost model. The extensible query optimizer uses the resulting estimations to decide upon the usability of the RI-tree for specific queries. Figure 12 presents the relative error of the estimated cost for the UNI and REAL databases. The relative errors stay below 14% and 23%, respectively. Figure 13 compares the absolute estimations and the actual cost for the blocked output of results ($output_{I/O}$). In addition, $join_{I/O}$ denotes the overhead due to the nested-loop join with the transient query tables. For the sake of comparability to the analytical I/O complexity mentioned in Section 3.3, the results are shown with respect to the actual query selectivity. Our interpretation of these results is twofold: First, the real I/O cost show that the total I/O is largely determined by the cardinality of the query result, whereas the overhead for the join processing remains almost constant. The relative cost of the join overhead decreases from 100% at 0% selectivity to 0.2‰ at 100% selectivity (Figure 13a). According to these empirical results, the overhead of $join_{I/O}$ is negligible for higher values of the query selectivity. Second, we observe that our cost model not only yields tight estimations for the total query cost, but also reflects the distribution between the output and join cost rather accurately. Regardless of the actual query selectivity, the cost computation on the databases UNI and REAL took about 0.05 seconds for a single interval intersection query.

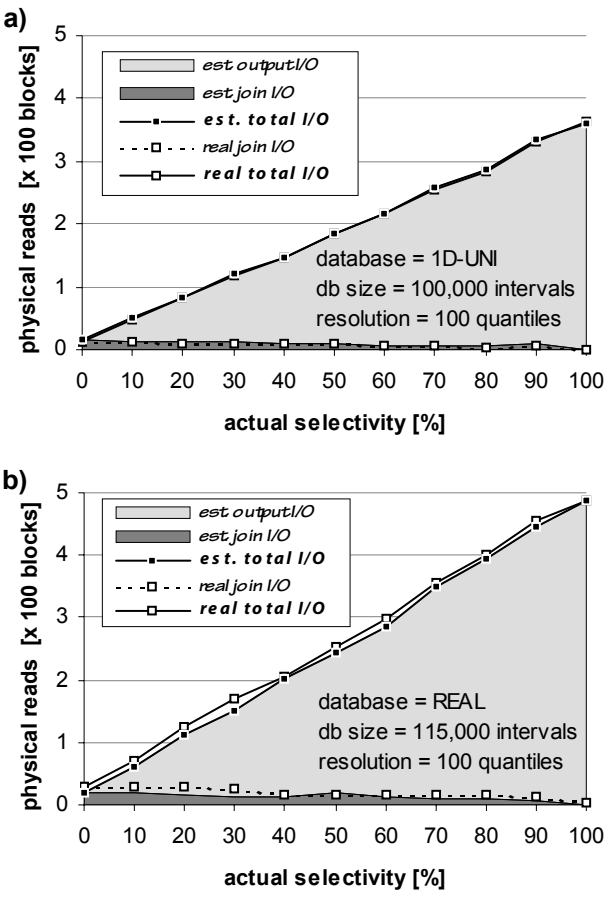

Fig. 13. Output cost and join overhead for queries evaluated **a)** on uniform data, **b)** on real data.

7. Conclusions

High quality selectivity estimation and cost prediction are the fundamentals of effective query optimization. Particularly for complex query objects and complex query predicates, the recent object-relational database servers provide extensible optimization frameworks that go along with the extensible indexing frameworks, in order to complete

the seamless integration of user-defined index structures into the declarative DML. In this paper, we present an example for such an extension and focus to the relational interval tree (RI-tree) that already fits well to modern object-relational extensible indexing frameworks. We particularly propose models to estimate the selectivity of interval intersection queries and to predict the cost for query processing. With respect to the generation and management of statistics, the proposed quantile-based selectivity estimation reuses as much built-in functionality of the RDBMS as possible. According to our experimental evaluation, the computed estimations are very accurate. For highly selective queries on a database of real interval data, the relative error for the selectivity estimation was around 2.9%. The corresponding errors for the I/O cost model amount to 23% and 3.3%, respectively.

In our future work, we plan to adapt the proposed techniques to support general interval relationships as required for temporal applications [18], and, in addition, to spatial queries in GIS and CAD applications [19].

References

[1] Acharya S., Poosala V., Ramaswamy S.: *Selectivity Estimation in Spatial Databases*. Proc. ACM SIGMOD, 13-24, 1999.

[2] Belussi A., Faloutsos C.: *Estimating the Selectivity of Spatial Queries Using the 'Correlation' Fractal Dimension*. Proc. VLDB, 299-310, 1995.

[3] Böhm C., Klump G., Kriegel H.-P.: *XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension*. Proc. SSD (LNCS 1651), 75-90, 1999.

[4] Bozkaya T., Özsoyoglu Z. M.: *Indexing Valid Time Intervals*. Proc. DEXA, LNCS 1460, 541-550, 1998.

[5] Boulos J., Ono K.: *Cost Estimation of User-Defined Methods in Object-Relational Database Systems*. ACM SIGMOD Record, 28(3), 22-28, 1999.

[6] Bliujute R., Saltenis S., Slivinskas G., Jensen C.S.: *Developing a DataBlade for a New Index*. Proc. ICDE, 314-323, 1999.

[7] Chen W., Chow J.-H., Fuh Y.-C., Grandbois J., Jou M., Mattos N., Tran B., Wang Y.: *High Level Indexing of User-Defined Types*. Proc. VLDB, 554-564, 1999.

[8] Edelsbrunner H.: *Dynamic Rectangle Intersection Searching*. Inst. for Information Processing Report 47, Technical University of Graz, Austria, 1980.

[9] Faloutsos C., Roseman S.: *Fractals for Secondary Key Retrieval*. Proc. ACM PODS, 247-252, 1989.

[10] Gaede V., Günther O.: *Multidimensional Access Methods*. ACM Computing Surveys 30(2), 170-231, 1998.

[11] Huang Y.-W., Jing N., Rundensteiner E. A.: A Cost Model for Estimating the Performance of Spatial Joins Using R-trees. Proc. SSDBM, 30-38, 1997.

[12] Haas P. J., Naughton J. F., Seshadri S., Stokes L.: *Sampling-Based Estimation of the Number of Distinct Values of an Attribute*. Proc. VLDB, 311-322, 1995.

[13] Hellerstein J., Stonebraker M.: *Predicate Migration: Optimizing Queries with Expensive Predicates*. Proc. ACM SIGMOD, 267-276, 1993.

[14] IBM Corp.: *IBM DB2 Universal Database Application Development Guide, Version 6*. Armonk, NY, 1999.

[15] Informix Software, Inc.: *DataBlade Developers Kit User's Guide, Version 3.4*. Menlo Park, CA, 1998.

[16] Kamel I., Faloutsos C.: *On Packing R-trees*. Proc. ACM CIKM, 490-499, 1993.

[17] Kriegel H.-P., Pötke M., Seidl T.: *Managing Intervals Efficiently in Object-Relational Databases*. Proc. VLDB, 407-418, 2000.

[18] Kriegel H.-P., Pötke M., Seidl T.: *Object-Relational Indexing for General Interval Relationships*. Proc. SSTD (LNCS 2121), 522-542, 2001.

[19] Kriegel H.-P., Pötke M., Seidl T.: *Interval Sequences: An Object-Relational Approach to Manage Spatial and Temporal Data*. Proc. SSTD (LNCS 2121), 481-501, 2001.

[20] Leutenegger S. T., Lopez M. A.: *The Effect of Buffering on the Performance of R-Trees*. Proc. ICDE, 164-171, 1998.

[21] Lipton R. J., Naughton J. F., Schneider A. D.: *Practical Selectivity Estimation through Adaptive Sampling*. Proc. ACM SIGMOD, 1-11, 1990.

[22] Lomet D.: *B-tree Page Size When Caching is Considered*. ACM SIGMOD Record 27(3), 28-32, 1998.

[23] Oracle Corp.: *Oracle8i Data Cartridge Developer's Guide, Release 2 (8.1.6)*. Redwood Shores, CA, 1999.

[24] Proietti G., Faloutsos C.: *Selectivity Estimation of Window Queries for Line Segment Datasets*. Proc. ACM CIKM, 340-347, 1998.

[25] Proietti G., Faloutsos C.: I/O *Complexity for Range Queries on Region Data Stored Using an R-tree*. Proc. ICDE, 628-635, 1999.

[26] Ramaswamy S.: *Efficient Indexing for Constraint and Temporal Databases*. Proc. ICDT (LNCS 1186), 419-431, 1997.

[27] Selinger P. G., Astrahan M. M., Chamberlin D. D., Lorie R. A., Price T. G.: *Access Path Selection in a Relational Database Management System*. Proc. ACM SIGMOD, 23-34, 1979.

[28] Slivinskas G., Jensen C. S., Snodgrass R. T.: Adaptable Query Optimization and Evaluation in Temporal Middleware. Proc. ACM SIGMOD, 127-138, 2001.

[29] Srinivasan J., Murthy R., Sundara S., Agarwal N., DeFazio S.: *Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i*. Proc. ICDE, 91-100, 2000.

[30] Snodgrass R. T.: *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 2000.

[31] Shen H., Ooi B. C., Lu H.: *The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases*. Proc. ICDE, 274-281, 1994.

[32] Stonebraker M.: *Inclusion of New Types in Relational Data Base Systems*. Proc. ICDE, 262-269, 1986.

[33] Theodoridis Y., Stefanakis E., Sellis T.: *Efficient Cost Models for Spatial Queries Using R-Trees*. IEEE Trans. on Knowledge and Data Engineering (TKDE), 12(1), 19-32, 2000.